Scientific computing using Cython: Best of both worlds!

March 18, 2017

Presenter: Simmi Mourya
Portland State University

Demo
github.com/simmimourya1/fossasia_17

Motivation

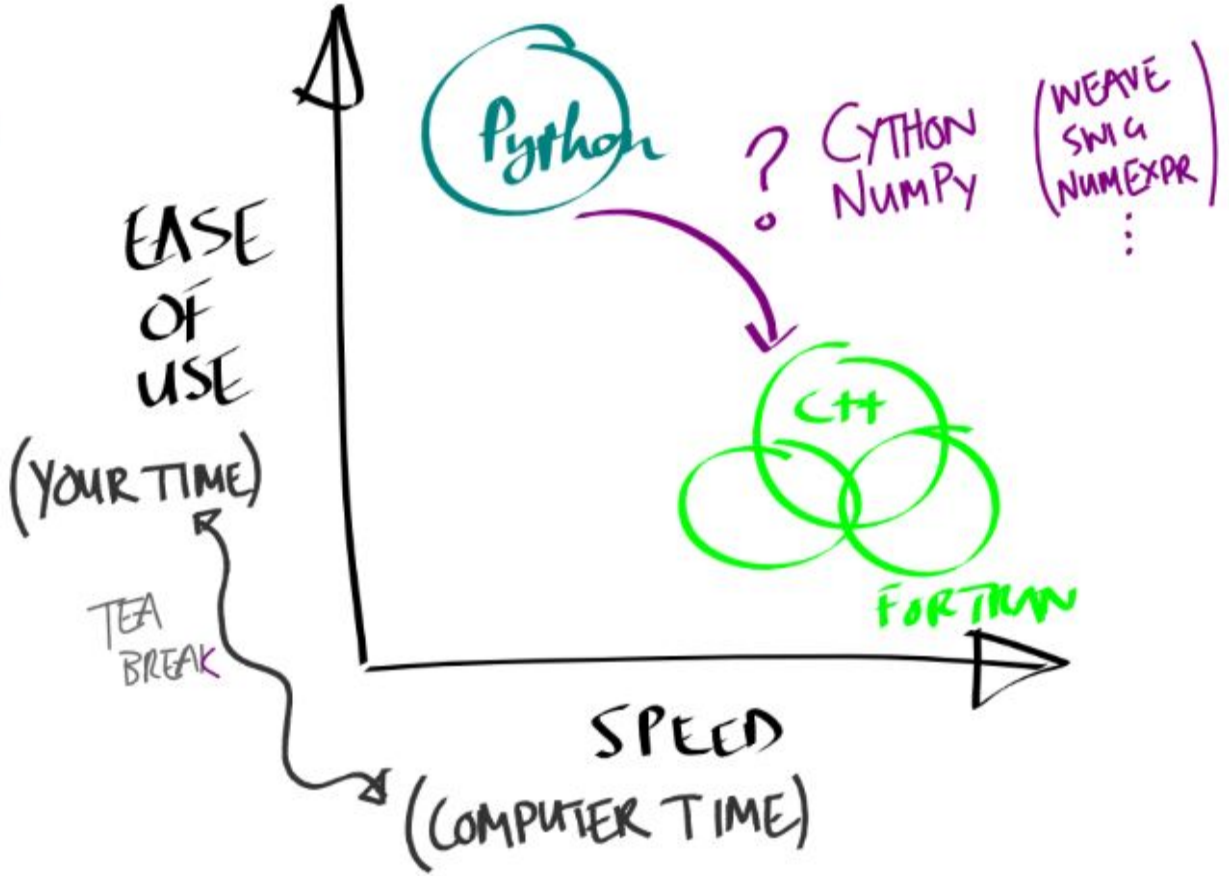


Illustration borrowed from: Stéfan van der Walt's presentation at Advanced Python Summer School, Kiel 2012.

Cython by example

Python 1x

```
def fib(n):
    a,b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

C/C++ 100x

```
int fib(int n)
{
    int tmp, i, a, b;
    a = b = 1;
    for(i=0; i<n; i++){
    tmp = a;
    a += b;
    b = tmp;
    }
}
```

Cython 80x

```
def fib(int n):
    cdef int i, a, b
    a, b = 1,1
    for i in range(n):
        a, b = a+b, a
    return a
```

Cython in wild

Project	Cython files	Cython SLOC
sage	761	477,000
numpy	14	5,000
scipy	28	24,000
pandas	21	27,000
lxml	12	22,000
scikits-learn	35	15,000
scikits-image	48	11,000
mpi4py	48	12,000
yt	45	18,000

Projects master branches as of November 2014

Cython at Glance

Cython is used for compiling Python-like code to machine-code

- Supports a big subset of the Python language
- Conditions and loops run 2-8x faster, overall 30% faster for plain Python code (vs. Py2.5, using Py Bench)

Cython at Glance

In addition:

- Add types for speedups (hundreds of times)
- Easily use native libraries (C/C++/Fortran) directly

How it works:

- Cython code is turned into C code which uses the CPython API and runtime

Coding in Cython is like coding in Python and C at the same time!

Use Case 1: Library Wrapping

- Cython is a popular choice for writing Python interface modules for C libraries

Use Case 2: Performance-critical code

- Python
- High-level
- Slow
- No variables typed

- C/C++/Fortran
- Lower-level
- Fast
- All variables typed

Common procedure: Where speed is needed, use a compiled language, then wrap the code for use from Python.

Breaking out of the global interpreter lock

- As a few of you might know, C Python has an infamous Global Interpreter Lock (GIL)

```
>>> import that
The Unwritten Rules of Python
1. You do not talk about the GIL.
2. You do NOT talk about the GIL.
3. Don't even mention the GIL. No seriously. …
```

- It limits thread performance

Watch David Beazley' "Understanding the Python GIL" lecture.

Consider this simple script, that runs twice, sequentially, a busy_sleep, i.e. a function that simulate a CPU intensive task:

```
def busy_sleep( ):
   while n > 0:
       n -= 1
N = 99999999
busy_sleep(N)
      busy_sleep(N)
```

This takes 6.7 seconds to run.

Now consider the threaded version:

```
from threading import Thread
def busy_sleep(n):
   while n > 0:
       n -= 1
N = 99999999
t1 = Thread(target=busy_sleep, args=(N, ))
t2 = Thread(target=busy_sleep, args=(N, ))
t1.start()
t2.start()
t1.join()
t2.join()
```

This takes 11.1 seconds to run. What!?

Cython offers a wonderful context manager to run instructions without the GIL: *with nogil*.

Demo

Building Cython code

Building Cython code

Cython code must, unlike Python, be compiled. This happens in two stages:

- A .pyx file is compiled by Cython to a .c file, containing the code of a Python extension module
- The .c file is compiled by a C compiler into a .so file (or .pyd on Windows) which can be imported directly into a Python session.

Building Cython code

There are several ways to build Cython code:

- Write a distutils setup.py.
- Use pyximport, importing Cython .pyx files as if they were .py files (using distutils to compile and build in the background).
- Use the Jupyter notebook or the Sage notebook, both of which allow Cython code inline.

Building Cython modules using distutils

Imagine a simple “hello world” script in a file hello.pyx:

```
def say_hello_to(name):
    print("Hello %s!" % name)
```

The following could be a corresponding setup.py script:

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
  name = 'Hello world app',
  ext_modules = cythonize("hello.pyx"),
)
```

Building Cython modules using distutils

To use this to build your Cython file use the command line options:

```
$ python setup.py build_ext --inplace
```

Which will leave a file in your local directory called helloworld.so (unix) or helloworld.pyd (Windows). Now to use this file: start the python interpreter and simply import it as if it was a regular python module:

```
>>> import helloworld
Hello World
```

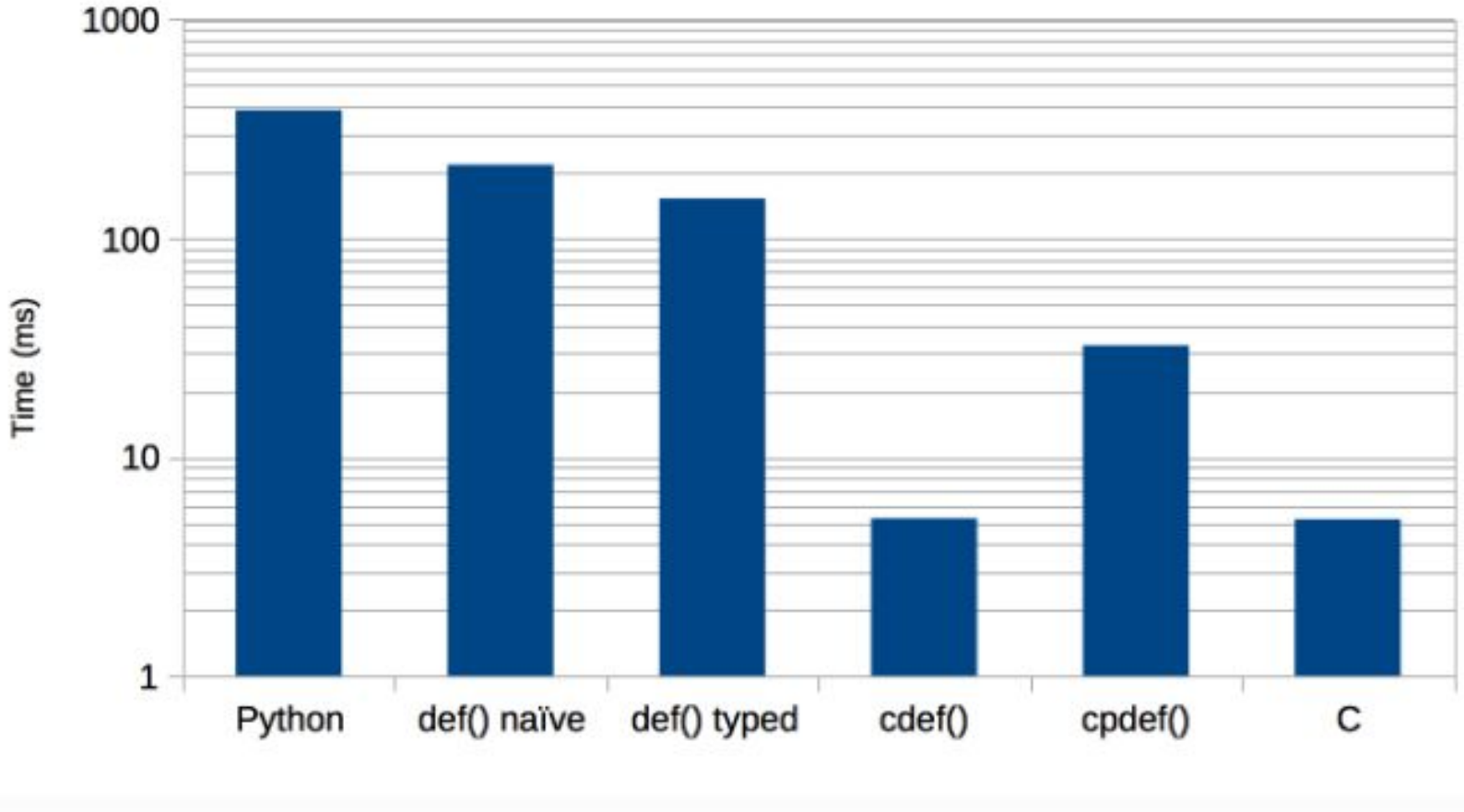
pyximport: Cython Compilation the Easy Way

To load .pyx files directly on import, without having to write a setup.py file.

```
>>> import pyximport; pyximport.install()
>>> import hello
Hello World
```

Demo: Building Cython modules using Jupyter notebook.

Language	Function call	Time (ms)	Speed, Python = 1
Python	`Fibo.fib(30)`	390	x 1
Cython	`cyFibo.fib(30)`	215	x 1.8
Cython	`cyFibo.fib_int(30)`	154	x 2.5
Cython	`cyFibo.fib_cdef(30)`	5.38	x72
Cython	`cyFibo.fib_cpdef(30)`	32.5	x12
C	`cFibo.fib(30)`	5.31	x73


1000
100
10
1
Time (ms)
Python
def() naïve
def() typed
cdef()
cpdef()
C

The conclusions that I draw from this are:

1. Naive Cython does speed things up, but not by much (x1.8).
2. Optimised Cython is fairly effortless (in this case) and worthwhile (x2.5).
3. cdef is really valuable (x72).
4. cpdef gives a good improvement over def because the recursive case exploits C functions.
5. Cython's cdef is insignificantly different from the more complicated C extension that is our best attempt.

NumPy and Cython

- Cython provides fast access to NumPy arrays

MemoryViews

Demo

Project demo

Cyvlfeat

Google summer of code 2016

@ Portland State University

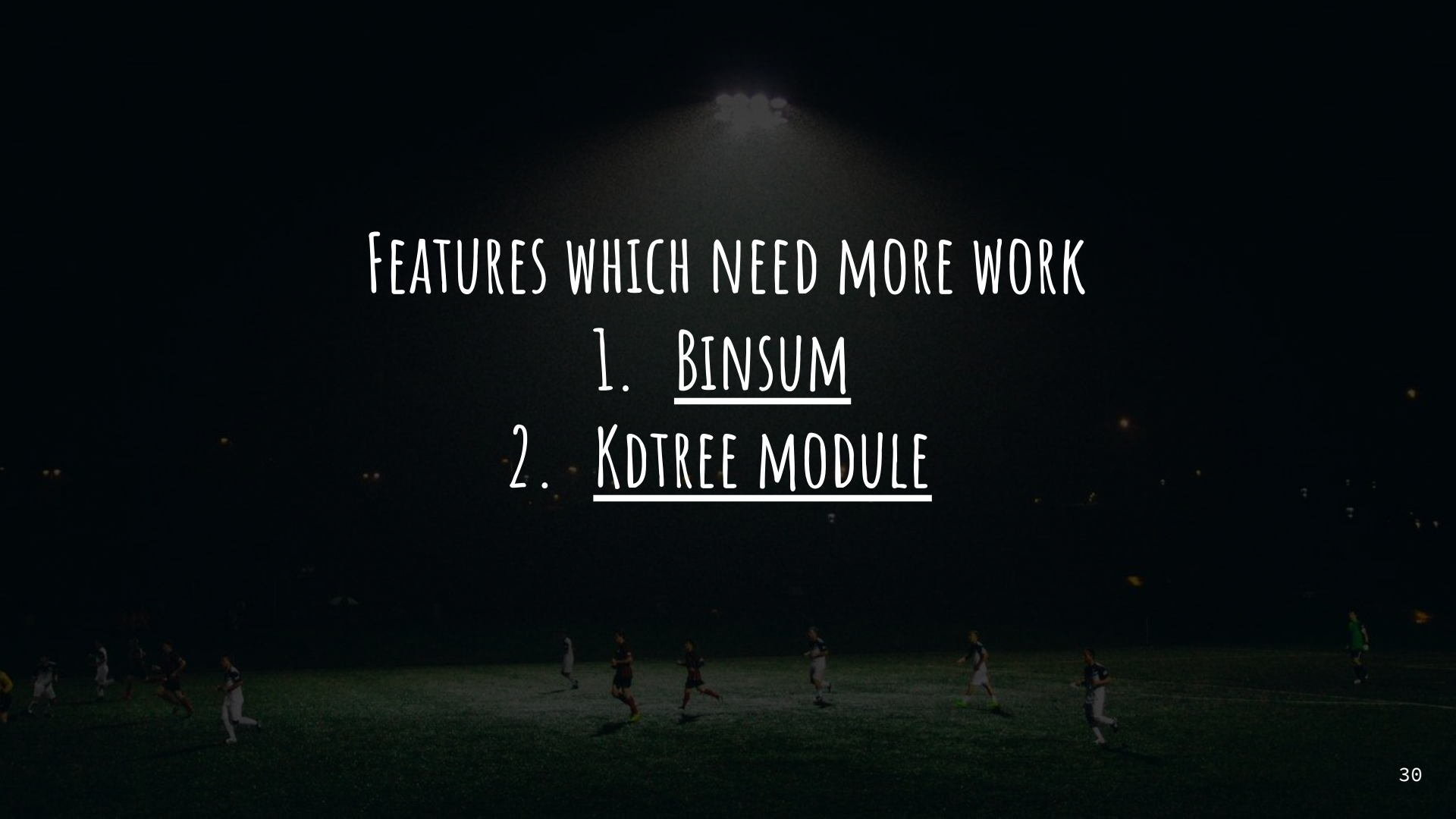
A cython/python wrapper for VLFeat

What is Vlfeat?

What is CYVLFEAT?

github.com/menpo/cyvlfeat


Contributions are welcome!
github.com/simmimouryal/cyvlfeat



Features which need more work

1. Binsum
2. Kdtree module

Questions ?

Thank you